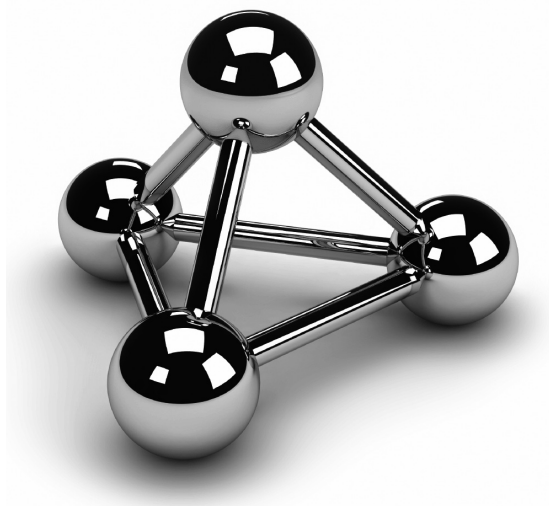
# Part II

# Transact-SQL Language

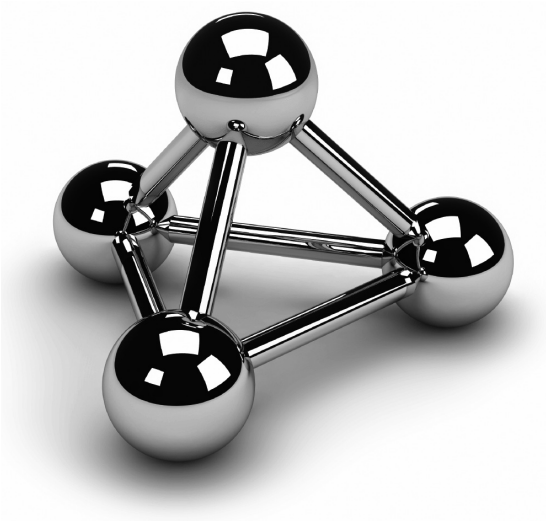


# Chapter 4

# SQL Components

## In This Chapter

- SQL's Basic Objects
- Data Types
- Transact-SQL Functions
- Scalar Operators
- NULL Values



This chapter introduces the elementary objects and basic operators supported by the Transact-SQL language. First, the basic language elements, including constants, identifiers, and delimiters, are described. Then, because every elementary object has a corresponding data type, data types are discussed in detail. Additionally, all existing operators and functions are explained. At the end of the chapter, NULL values are introduced.

# SQL's Basic Objects

The language of Database Engine, Transact-SQL, has the same basic features as other common programming languages:

- Literal values (also called constants)
- Delimiters
- Comments
- Identifiers
- Reserved keywords

The following sections describe these features.

## Literal Values

A *literal* value is an alphanumerical, hexadecimal, or numeric constant. A string constant contains one or more characters of the character set enclosed in two single straight quotation marks (' ') or double straight quotation marks (" ") (single quotation marks are preferred due to the multiple uses of double quotation marks, as discussed in a moment). If you want to include a single quotation mark within a string delimited by single quotation marks, use two consecutive single quotation marks within the string. Hexadecimal constants are used to represent nonprintable characters and other binary data. Each hexadecimal constant begins with the characters '0x' followed by an even number of characters or numbers. Examples 4.1 and 4.2 illustrate some valid and invalid string constants and hexadecimal constants.

**EXAMPLE 4.1**

Some valid string constants and hexadecimal constants follow:

'Philadelphia'
"Berkeley, CA 94710"
'9876'
'Apostrophe is displayed like this: can''t' (note the two consecutive single quotation marks)
0x53514C0D

**EXAMPLE 4.2**

The following are *not* string constants:

'AB'C' (odd number of single quotation marks)
'New York" (same type of quotation mark—single or double—must be used at each end of the string)

The numeric constants include all integer, fixed-point, and floating-point values with and without signs (see Example 4.3).

**EXAMPLE 4.3**

The following are numeric constants:

130
–130.00
–0.357E5 (scientific notation—*n*E*m* means *n* multiplied by $10^m$)
22.3E-3

A constant always has a data type and a length, and both depend on the format of the constant. Additionally, every numeric constant has a precision and a scale factor. (The data types of the different kinds of literal values are explained later in this chapter.)

## Delimiters

In Transact-SQL, double quotation marks have two meanings. In addition to enclosing strings, double quotation marks can also be used as delimiters for so-called *delimited identifiers*. Delimited identifiers are a special kind of identifier usually used to allow the use of reserved keywords as identifiers and also to allow spaces in the names of database objects.

**NOTE**

*Differentiation between single and double quotation marks was first introduced in the SQL92 standard. In the case of identifiers, this standard differentiates between regular and delimited identifiers. Two key differences are that delimited identifiers are enclosed in double quotation marks and are case sensitive. (Transact-SQL also supports the use of square brackets instead of double quotation marks.) Double quotation marks are only used for delimiting strings. Generally, delimited identifiers were introduced to allow the specification of identifiers, which are otherwise identical to reserved keywords. Specifically, delimited identifiers protect you from using names (identifiers, variable names) that could be introduced as reserved keywords in one of the future SQL standards. Also, delimited identifiers may contain characters that are normally illegal within identifier names, such as blanks.*

In Transact-SQL, the use of double quotation marks is defined using the QUOTED_IDENTIFIER option of the SET statement. If this option is set to ON, which is the default value, an identifier in double quotation marks will be defined as a delimited identifier. In this case, double quotation marks cannot be used for delimiting strings.

## Comments

There are two different ways to specify a comment in a Transact-SQL statement. Using the pair of characters /* and */ marks the enclosed text as a comment. In this case, the comment may extend over several lines. Furthermore, the characters -- (two hyphens) indicate that the remainder of the current line is a comment. (The two hyphens -- comply with the ANSI SQL standard, while /* and */ are the extensions of Transact-SQL.)

## Identifiers

In Transact-SQL, identifiers are used to identify database objects such as databases, tables, and indices. They are represented by character strings that may include up to 128 characters and can contain letters, numerals, or the following characters: _, @, #, and $. Each name must begin with a letter or one of the following characters: _, @, or #. The character # at the beginning of a table or stored procedure name denotes a temporary object, while @ at the beginning of a name denotes a variable. As indicated earlier, these rules don't apply to delimited identifiers (also known as quoted identifiers), which can contain, or begin with, any character (other than the delimiters themselves).

## Reserved Keywords

Each programming language has a set of names with reserved meanings, which must be written and used in the defined format. Names of this kind are called *reserved keywords*. Transact-SQL uses a variety of such names, which, as in many other programming languages, cannot be used as object names, unless the objects are specified as delimited or quoted identifiers.

**NOTE**

*In Transact-SQL, the names of all data types and system functions, such as CHARACTER and INTEGER, are not reserved keywords. They can therefore be used for denoting objects. (Do not use data types and system functions as object names! Such a use makes Transact-SQL statements difficult to read and understand.)*

# Data Types

All the data values of a column must be of the same data type. (The only exception specifies the values of the SQL_VARIANT data type.) Transact-SQL uses different data types, which can be categorized as follows:

- Numeric data types
- Character data types
- Temporal (date and/or time) data types
- Miscellaneous data types
- DECIMAL with VARDECIMAL storage type

The following sections describe all these categories.

## Numeric Data Types

Numeric data types are used to represent numbers. The following table shows the list of all numeric data types:

| Data Type | Explanation |
|---|---|
| INTEGER | Represents integer values that can be stored in 4 bytes. The range of values is –2,147,483,648 to 2,147,483,647. INT is the short form for INTEGER. |
| SMALLINT | Represents integer values that can be stored in 2 bytes. The range of values is –32768 to 32767. |
| TINYINT | Represents nonnegative integer values that can be stored in 1 byte. The range of values is 0 to 255. |
| BIGINT | Represents integer values that can be stored in 8 bytes. The range of values is $-2^{63}$ to $2^{63} - 1$. |
| DECIMAL(p,[s]) | Describes fixed-point values. The argument **p** (precision) specifies the total number of digits with assumed decimal point **s** (scale) digits from the right. DECIMAL values are stored, depending on the value of **p**, in 5 to 17 bytes. DEC is the short form for DECIMAL. |
| NUMERIC(p,[s]) | Synonym for DECIMAL. |
| REAL | Used for floating-point values. The range of positive values is approximately 2.23E – 308 to 1.79E + 308, and the range of negative values is approximately –1.18E – 38 to –1.18E + 38 (the value zero can also be stored). |
| FLOAT[(p)] | Represents floating-point values, like REAL. **p** defines the precision with **p** < 25 as single precision (4 byte) and **p** >= 25 as double precision (8 byte). |
| MONEY | Used for representing monetary values. MONEY values correspond to 8-byte DECIMAL values and are rounded to four digits after the decimal point. |
| SMALLMONEY | Corresponds to the data type MONEY but is stored in 4 bytes. |

## Character Data Types

There are two general forms of character data types. They can be strings of single-byte characters or strings of Unicode characters. (Unicode uses several bytes to specify one character.) Further, strings can have fixed or variable length. The following character data types are used:

| Data Type | Explanation |
|---|---|
| CHAR[(n)] | Represents a fixed-length string of single-byte characters, where **n** is the number of characters inside the string. The maximum value of **n** is 8000. CHARACTER(n) is an additional equivalent form for CHAR(n). If **n** is omitted, the length of the string is assumed to be 1. |
| VARCHAR[(n)] | Describes a variable-length string of single-byte characters ($0 < \mathbf{n} \leq 8000$). In contrast to the CHAR data type, the values for the VARCHAR data type are stored in their actual length. This data type has two synonyms: CHAR VARYING and CHARACTER VARYING. |
| NCHAR[(n)] | Stores fixed-length strings of Unicode characters. The main difference between the CHAR and NCHAR data types is that each character of the NCHAR data type is stored in 2 bytes, while each character of the CHAR data type uses 1 byte of storage space. Therefore, the maximum number of characters in a column of the NCHAR data type is 4000. |
| NVARCHAR[(n)] | Stores variable-length strings of Unicode characters. The main difference between the VARCHAR and the NVARCHAR data types is that each NVARCHAR character is stored in 2 bytes, while each VARCHAR character uses 1 byte of storage space. The maximum number of characters in a column of the NVARCHAR data type is 4000. |

**NOTE**

*The VARCHAR data type is identical to the CHAR data type except for one difference: if the content of a CHAR(n) string is shorter than **n** characters, the rest of the string is padded with blanks. (A value of the VARCHAR data type is always stored in its actual length.)*

## Temporal Data Types

Transact-SQL supports the following temporal data types:

- DATETIME
- SMALLDATETIME
- DATE
- TIME
- DATETIME2
- DATETIMEOFFSET

The DATETIME and SMALLDATETIME data types specify a date and time, with each value being stored as an integer value in 4 bytes or 2 bytes, respectively. Values of DATETIME and SMALLDATETIME are stored internally as two separate numeric values. The date value of DATETIME is stored in the range 01/01/1753 to 12/31/9999. The analog value of SMALLDATETIME is stored in the range 01/01/1900 to 06/06/2079. The time component is stored in the second 4-byte (or 2-byte for SMALLDATETIME) field as the number of three-hundredths of a second (DATETIME) or minutes (SMALLDATETIME) that have passed since midnight.

The use of DATETIME and SMALLDATETIME is rather inconvenient if you want to store only the date part or time part. For this reason, SQL Server 2008 introduces the new data types DATE and TIME, which store just the DATE or TIME component of a DATETIME, respectively. The DATE data type is stored in 3 bytes and has the range 01/01/0001 to 12/31/9999. The TIME data type is stored in 3–5 bytes and has an accuracy of 100 nanoseconds (ns).

The DATETIME2 data type is also a new data type that stores high-precision date and time data. The data type can be defined for variable lengths depending on the requirement. (The storage size is 6–8 bytes). The accuracy of the time part is 100 ns. This data type isn't aware of Daylight Saving Time.

All the temporal data types described thus far don't have support for the time zone. The new data type called DATETIMEOFFSET has the time zone offset portion. For this reason, it is stored in 6–8 bytes. (All other properties of this data type are analogous to the corresponding properties of DATETIME2.)

The date value in Transact-SQL is by default specified as a string in a format like 'mmm dd yyyy' (e.g., 'Jan 10 1993') inside two single quotation marks or double quotation marks. (Note that the relative order of month, day, and year can be controlled by the SET DATEFORMAT statement. Additionally, the system recognizes numeric month values with delimiters of / or –.) Similarly, the time value is specified in the format 'hh:mm' and Database Engine uses 24-hour time (23:24, for instance).

**NOTE**

*Transact-SQL supports a variety of input formats for DATETIME values. As you already know, both objects are identified separately; thus, date and time values can be specified in any order or alone. If one of the values is omitted, the system uses the default values. (The default value for time is 12:00 AM.)*

Examples 4.4 and 4.5 show the different ways, how date or time values can be written using the different formats.

**EXAMPLE 4.4**

The following date descriptions can be used:

'28/5/1959' (with SET DATEFORMAT dmy)
'May 28, 1959'
'1959 MAY 28'

**EXAMPLE 4.5**

The following time expressions can be used:

'8:45 AM'
'4 pm'

## Miscellaneous Data Types

Transact-SQL supports several data types that do not belong to any of the data type groups described previously:

- Binary data types
- BIT
- Large object data types
- CURSOR (discussed in Chapter 8)
- UNIQUEIDENTIFIER
- SQL_VARIANT
- TABLE (discussed in Chapter 8)
- XML (discussed in Chapter 28)
- Spatial (e.g., GEOGRAPHY and GEOMETRY) data types (discussed in Chapter 29 )
- HIERARCHYID
- TIMESTAMP data type
- User-defined data types (discussed in Chapter 5)

The following sections describe each of these data types (other than those designated as being discussed in another chapter).

### Binary and BIT Data Types

BINARY and VARBINARY are the two binary data types. They describe data objects being represented in the internal format of the system. They are used to store bit strings. For this reason, the values are entered using hexadecimal numbers.

The values of the BIT data type are stored in a single bit. Therefore, up to 8 bit columns are stored in 1 byte. The following table summarizes the properties of these data types:

| Data Type | Explanation |
|---|---|
| BINARY[(n)] | Specifies a bit string of fixed length with exactly **n** bytes (0 < **n** ≤ 8000). |
| VARBINARY[(n)] | Specifies a bit string of variable length with up to **n** bytes (0 < **n** ≤ 8000). |
| BIT | Used for specifying the Boolean data type with three possible values: FALSE, TRUE, and NULL. |

## Large Object Data Types

Large objects (LOBs) are data objects with the maximum length of 2GB. These objects are generally used to store large text data and to load modules and audio/video files. Transact-SQL supports two different ways to specify and access LOBs:

- Use the data types VARCHAR(MAX), NVARCHAR(MAX), and VARBINARY(MAX)
- Use the so-called text/image data type

The following subsections describe the two forms of LOBs.

**The MAX Specifier** Starting with SQL Server 2005, you can use the same programming model to access values of standard data types and LOBs. In other words, you can use convenient system functions and string operators to work with LOBs.

Database Engine uses the MAX specifier with the data types VARCHAR, NVARCHAR, and VARBINARY to define variable-length columns. When you use MAX by default (instead of an explicit value), the system analyzes the length of the particular string and decides whether to store the string as a convenient value or as a LOB. The MAX specifier indicates that the size of column values can reach the maximum LOB size of the current system. (In a future version of SQL Server, it is possible that MAX will have a higher maximum value for strings.)

Although the database system decides how a LOB will be stored, you can override this default specification using the **sp_tableoption** system procedure with the LARGE_VALUE_TYPES_OUT_OF_ROW option. If the option's value is set to 1, the data in columns declared using the MAX specifier will be stored separately from all other data. If this option is set to 0, Database Engine stores all values for the row size < 8060 bytes as regular row data.

In SQL Server 2008, you can apply the new FILESTREAM attribute to a VARBINARY(MAX) column to store large binary data directly in an NTFS file system. The main advantage of this attribute is that the size of the corresponding LOB is limited only by the file system volume size.

**TEXT, NTEXT, and IMAGE Data Types** The data types TEXT, NTEXT, and IMAGE constitute the so-called text/image data types. Data objects of the type IMAGE can contain any kind of data (load modules, audio/video), while data objects of the data types TEXT and NTEXT can contain any text data (that is, printable data).

The text/image data types are stored by default separately from all other values of a database using a B-tree structure that points to the fragments of that data. (A *B-tree structure* is a treelike data structure in which all of the bottom nodes are the same number of levels away from the root of the tree.) For columns of a text/image data type, Database Engine stores a 16-byte pointer in the data row that specifies where the data can be found.

If the amount of text/image data is less than 32KB, the pointer points to the root node of the B-tree structure, which is 84 bytes long. The root node points to the physical blocks of the data. If the amount of the data is greater than 32KB, Database Engine builds intermediate nodes between the data blocks and the root node.

For each table that contains more than one column with such data, all values of the columns are stored together. However, one physical page can hold only text/image data from a single table.

Although text/image data is stored separately from all other data, you can modify this using the **sp_tableoption** system procedure with the TEXT_IN_ROW option. Using this option, you can specify the maximum number of bytes, which are stored together with the regular data.

The text/image data types discussed this far have several limitations. You can't use them as local variables (in stored procedures or in groups of Transact-SQL statements). Also, they can't be a part of an index or can't be used in the following clauses of the SELECT statement: WHERE, ORDER BY, and GROUP BY. The biggest problem concerning all text/image data types is that you have to use special operators (READTEXT, WRITETEXT, and UPDATETEXT) to work with such data.

**NOTE**

*The text/image data types are marked as a deprecated feature and will be removed in a future version of Database Engine. Use VARCHAR(MAX), NVARCHAR(MAX), and VARBINARY(MAX) instead.*

## UNIQUEIDENTIFIER Data Type

As its name implies, a value of the UNIQUEIDENTIFIER data type is a unique identification number stored as a 16-byte binary string. This data type is closely related to the globally unique identifier (GUID), which guarantees uniqueness worldwide. Hence, using this data type, you can uniquely identify data and objects in distributed systems.

The initialization of a column or a variable of the UNIQUEIDENTIFIER type can be provided using the functions NEWID and NEWSEQUENTIALID, as well as with

a string constant written in a special form using hexadecimal digits and hyphens. (The functions NEWID and NEWSEQUENTIALID are described in the section "System Functions" later in this chapter.)

A column of the UNIQUEIDENTIFIER data type can be referenced using the keyword ROWGUIDCOL in a query to specify that the column contains ID values. (This keyword does not generate any values.) A table can have several columns of the UNIQUEIDENTIFIER type, but only one of them can have the ROWGUIDCOL keyword.

## SQL_VARIANT Data Type

The SQL_VARIANT data type can be used to store values of various data types at the same time, such as numeric values, strings, and date values. (The only types of values that cannot be stored are TIMESTAMP values.) Each value of an SQL_VARIANT column has two parts: the data value and the information that describes the value. (This information contains all properties of the actual data type of the value, such as length, scale, and precision.)

Transact-SQL supports the SQL_VARIANT_PROPERTY function, which displays the attached information for each value of an SQL_VARIANT column. For the use of the SQL_VARIANT data type, see Example 5.5 in Chapter 5.

**NOTE**

*Declare a column of a table using the SQL_VARIANT data type only if it is really necessary. A column should have this data type if its values may be of different types or if it is not possible to determine the type of a column during the database design process.*

## HIERARCHYID Data Type

The HIERARCHYID data type is used to store an entire hierarchy. It is implemented as a Common Language Runtime (CLR) user-defined type that comprises several system functions for creating and operating on hierarchy nodes. The following functions, among others, belong to the methods of this data type: **GetAncestor()**, **GetDescendant()**, **Read()**, and **Write()**. (The detailed description of this data type is outside the scope of this book.)

## TIMESTAMP Data Type

The TIMESTAMP data type specifies a column being defined as VARBINARY(8) or BINARY(8), depending on nullability of the column. The system maintains a current value (not a date or time) for each database, which it increments whenever any row with

a TIMESTAMP column is inserted or updated and sets the TIMESTAMP column to that value. Thus, TIMESTAMP columns can be used to determine the relative time when rows were last changed. (ROWVERSION is a synonym for TIMESTAMP.)

**NOTE**

*The value stored in a TIMESTAMP column isn't important by itself. This column is usually used to detect whether a specific row has been changed since the last time it was accessed.*

## DECIMAL with VARDECIMAL Storage Format

The DECIMAL data type is typically stored on the disk as fixed-length data. Since SQL Server 2005 SP2, this data type can be stored as a variable-length column by using the new storage format called VARDECIMAL. Using VARDECIMAL, you can significantly reduce the storage space for a DECIMAL column in which values have significant difference in their lengths.

**NOTE**

*VARDECIMAL is a storage format and not a data type.*

The VARDECIMAL storage format is helpful when you have to specify the largest possible value for a field in which the stored values usually are much smaller. Table 4-1 shows this.

**NOTE**

*The VARDECIMAL storage format works the same way for the DECIMAL data type as the VARCHAR data type works for alphanumerical data.*

| Precision | No. of Bytes: VARDECIMAL | No. of Bytes: Fixed Length |
|---|---|---|
| 0 or NULL | 2 | 5 |
| 1 | 4 | 5 |
| 20 | 12 | 13 |
| 30 | 16 | 17 |
| 38 | 20 | 17 |

**Table 4-1** *Number of Bytes for Storing VARDECIMAL and Fixed Length*

To enable the VARDECIMAL storage format, you first have to enable it for the database and then enable it for the particular table of that database. The **sp_db_vardecimal_storage_format** system procedure is used for the first step, as Example 4.6 shows.

**EXAMPLE 4.6**

```
EXEC sp_db_vardecimal_storage_format 'sample', 'ON';
```

The VARDECIMAL STORAGE FORMAT option of the **sp_table option** system procedure is used to turn on this storage for the table. Example 4.7 turns on the VARDECIMAL storage format for the **project** table.

**EXAMPLE 4.7**

```
EXEC sp_tableoption 'project', 'vardecimal storage format', 1
```

As you already know, the main reason to use the VARDECIMAL storage format is to reduce the storage size of the data. If you want to test how much storage space could be gained by using this storage format, use the dynamic management view called **sys.sp_estimated_rowsize_reduction_for_vardecimal**. This dynamic management view gives you a detailed estimate for the particular table.

# Transact-SQL Functions

Transact-SQL functions can be either aggregate functions or scalar functions. The following sections describe these function types.

## Aggregate Functions

*Aggregate functions* are applied to a group of data values from a column. Aggregate functions always return a single value. Transact-SQL supports several groups of aggregate functions:

- Convenient aggregate functions
- Statistical aggregate functions
- User-defined aggregate functions
- Analytic aggregate functions

Statistical and analytic aggregates are discussed in Chapter 24. User-defined aggregates are beyond the scope of this book. That leaves the convenient aggregate functions, described next:

- **AVG** Calculates the arithmetic mean (average) of the data values contained within a column. The column must contain numeric values.
- **MAX and MIN** Calculate the maximum and minimum data value of the column, respectively. The column can contain numeric, string, and date/time values.
- **SUM** Calculates the total of all data values in a column. The column must contain numeric values.
- **COUNT** Calculates the number of (non-null) data values in a column. The only aggregate function not being applied to columns is COUNT(*). This function returns the number of rows (whether or not particular columns have NULL values).
- **COUNT_BIG** Analogous to COUNT, the only difference being that COUNT_BIG returns a value of the BIGINT data type.

The use of convenient aggregate functions with the SELECT statement are described in detail in Chapter 6.

## Scalar Functions

In addition to aggregate functions, Transact-SQL provides several scalar functions that are used in the construction of scalar expressions. (A scalar function operates on a single value or list of values, as opposed to aggregate functions, which operate on the data from multiple rows.) Scalar functions can be categorized as follows:

- Numeric functions
- Date functions
- String functions
- System functions
- Metadata functions

The following sections describe these function types.

## Numeric Functions

Numeric functions within Transact-SQL are mathematical functions for modifying numeric values. The following numeric functions are available:

| Function | Explanation |
|---|---|
| ABS(n) | Returns the absolute value (i.e., negative values are returned as positive) of the numeric expression **n**. Example:<br>SELECT ABS(–5.767) = 5.767, SELECT ABS(6.384) = 6.384 |
| ACOS(n) | Calculates arc cosine of **n**. **n** and the resulting value belong to the FLOAT data type. |
| ASIN(n) | Calculates the arc sine of **n**. **n** and the resulting value belong to the FLOAT data type. |
| ATAN(n) | Calculates the arc tangent of **n**. **n** and the resulting value belong to the FLOAT data type. |
| ATN2(n,m) | Calculates the arc tangent of **n**/**m**. **n**, **m**, and the resulting value belong to the FLOAT data type. |
| CEILING(n) | Returns the smallest integer value greater or equal to the specified parameter. Examples:<br>SELECT CEILING(4.88) = 5<br>SELECT CEILING(–4.88) = –4 |
| COS(n) | Calculates the cosine of **n**. **n** and the resulting value belong to the FLOAT data type. |
| COT(n) | Calculates the cotangent of **n**. **n** and the resulting value belong to the FLOAT data type. |
| DEGREES(n) | Converts radians to degrees. Examples:<br>SELECT DEGREES(PI()/2) = 90.0<br>SELECT DEGREES(0.75) = 42.97 |
| EXP(n) | Calculates the value $\mathbf{e}^{\mathbf{n}}$. Example: SELECT EXP(1) = 2.7183 |
| FLOOR(n) | Calculates the largest integer value less than or equal to the specified value **n**. Example:<br>SELECT FLOOR(4.88) = 4 |
| LOG(n) | Calculates the natural (i.e., base e) logarithm of **n**. Examples:<br>SELECT LOG(4.67) = 1.54<br>SELECT LOG(0.12) = –2.12 |
| LOG10(n) | Calculates the logarithm (base 10) for **n**. Examples:<br>SELECT LOG10(4.67) = 0.67<br>SELECT LOG10(0.12) = –0.92 |
| PI() | Returns the value of the number pi (3.14). |
| POWER(x,y) | Calculates the value $\mathbf{x}^{\mathbf{y}}$. Examples:<br>SELECT POWER(3.12,5) = 295.65<br>SELECT POWER(81,0.5) = 9 |
| RADIANS(n) | Converts degrees to radians. Examples:<br>SELECT RADIANS(90.0) = 1.57<br>SELECT RADIANS(42.97) = 0.75 |
| RAND | Returns a random number between 0 and 1 with a FLOAT data type. |

| Function | Explanation |
|---|---|
| ROUND(n, p,[t]) | Rounds the value of the number **n** by using the precision **p**. Use positive values of **p** to round on the right side of the decimal point and use negative values to round on the left side. An optional parameter **t** causes **n** to be truncated. Examples:<br>SELECT ROUND(5.4567,3) = 5.4570<br>SELECT ROUND(345.4567,–1) = 350.0000<br>SELECT ROUND(345.4567,–1,1) = 340.0000 |
| ROWCOUNT_BIG | Returns the number of rows that have been affected by the last Transact-SQL statement executed by the system. The return value of this function has the BIGINT data type. |
| SIGN(n) | Returns the sign of the value **n** as a number (+1 for positive, –1 for negative, and 0 for zero). Example:<br>SELECT SIGN(0.88) = 1 |
| SIN(n) | Calculates the sine of **n**. **n** and the resulting value belong to the FLOAT data type. |
| SQRT(n) | Calculates the square root of **n**. Example:<br>SELECT SQRT(9) = 3 |
| SQUARE(n) | Returns the square of the given expression. Example:<br>SELECT SQUARE(9) = 81 |
| TAN(n) | Calculates the tangent of **n**. **n** and the resulting value belong to the FLOAT data type. |

## Date Functions

Date functions calculate the respective date or time portion of an expression or return the value from a time interval. Transact-SQL supports the following date functions:

| Function | Explanation |
|---|---|
| GETDATE() | Returns the current system date and time. Example:<br>SELECT GETDATE() = 2008-01-01 13:03:31.390 |
| DATEPART(item,date) | Returns the specified part **item** of a date **date** as an integer. Examples:<br>SELECT DATEPART(month, '01.01.2005') = 1 (1 = January)<br>SELECT DATEPART(weekday, '01.01.2005') = 7 (7 = Sunday) |
| DATENAME(item,date) | Returns the specified part **item** of the date **date** as a character string. Example:<br>SELECT DATENAME(weekday, '01.01.2005') = Saturday |
| DATEDIFF(item,dat1,dat2) | Calculates the difference between the two date parts **dat1** and **dat2** and returns the result as an integer in units specified by the value **item**. Example:<br>SELECT DATEDIFF(year, BirthDate, GETDATE()) AS age FROM employee; -> returns the age of each employee. |
| DATEADD(i,n,d) | Adds the number **n** of units specified by the value **i** to the given date **d**. Example:<br>SELECT DATEADD(DAY,3,HireDate) AS age FROM employee; -> adds three days to the starting date of employment of every employee (see the **sample** database). |

## String Functions

String functions are used to manipulate data values in a column, usually of a character data type. Transact-SQL supports the following string functions:

| Function | Explanation |
|---|---|
| ASCII(character) | Converts the specified character to the equivalent integer (ASCII) code. Returns an integer. Example:<br>SELECT ASCII('A') = 65 |
| CHAR(integer) | Converts the ASCII code to the equivalent character. Example:<br>SELECT CHAR(65) = 'A'. |
| CHARINDEX(z1,z2) | Returns the starting position where the partial string **z1** first occurs in the string **z2**. Returns 0 if **z1** does not occur in **z2**. Example:<br>SELECT CHARINDEX('bl', 'table') = 3. |
| DIFFERENCE(z1,z2) | Returns an integer, 0 through 4, that is the difference of SOUNDEX values of two strings **z1** and **z2**. (SOUNDEX returns a number that specifies the sound of a string. With this method, strings with similar sounds can be determined.) Example:<br>SELECT DIFFERENCE('spelling', 'telling') = 2 (sounds a little bit similar, 0 = doesn't sound similar) |
| LEFT(z, length) | Returns the first **length** characters from the string **z**. |
| LEN(z) | Returns the number of characters, instead of the number of bytes, of the specified string expression, excluding trailing blanks. |
| LOWER(z1) | Converts all uppercase letters of the string **z1** to lowercase letters. Lowercase letters and numbers, and other characters, do not change. Example:<br>SELECT LOWER('BiG') = 'big' |
| LTRIM(z) | Removes leading blanks in the string **z**. Example:<br>SELECT LTRIM('  String') = 'String' |
| NCHAR(i) | Returns the Unicode character with the specified integer code, as defined by the Unicode standard. |
| QUOTENAME(char_string) | Returns a Unicode string with the delimiters added to make the input string a valid delimited identifier. |
| PATINDEX(%p%,expr) | Returns the starting position of the first occurrence of a pattern **p** in a specified expression **expr**, or zeros if the pattern is not found. Examples:<br>1) SELECT PATINDEX('%gs%', 'longstring') = 4;<br>2) SELECT RIGHT(ContactName, LEN(ContactName)-PATINDEX('% %',ContactName)) AS First_name FROM Customers;<br>(The second query returns all first names from the **customers** column.) |
| REPLACE(str1,str2,str3) | Replaces all occurrences of the **str2** in the **str1** with the **str3**. Example:<br>SELECT REPLACE('shave' , 's' , 'be') = behave |

| Function | Explanation |
|---|---|
| REPLICATE(z,i) | Repeats string **z i** times. Example:<br>SELECT REPLICATE('a',10) = 'aaaaaaaaaa' |
| REVERSE(z) | Displays the string **z** in the reverse order. Example:<br>SELECT REVERSE('calculate') = 'etaluclac' |
| RIGHT(z,length) | Returns the last **length** characters from the string **z**. Example:<br>SELECT RIGHT('Notebook',4) = 'book' |
| RTRIM(z) | Removes trailing blanks of the string **z**. Example:<br>SELECT RTRIM('Notebook ') = 'Notebook' |
| SOUNDEX(a) | Returns a four-character SOUNDEX code to determine the similarity between two strings. Example:<br>SELECT SOUNDEX('spelling') = S145 |
| SPACE(length) | Returns a string with spaces of length specified by **length**. Example:<br>SELECT SPACE = ' ' |
| STR(f,[len [,d]]) | Converts the specified float expression **f** into a string. **len** is the length of the string including decimal point, sign, digits, and spaces (10 by default), and **d** is the number of digits to the right of the decimal point to be returned. Example:<br>SELECT STR(3.45678,4,2) = '3.46' |
| STUFF(z1,a,length,z2) | Replaces the partial string **z1** with the partial string **z2** starting at position a, replacing **length** characters of **z1**. Examples:<br>SELECT STUFF('Notebook',5,0, ' in a ') = 'Note in a book'<br>SELECT STUFF('Notebook',1,4, 'Hand') = 'Handbook' |
| SUBSTRING(z,a,length) | Creates a partial string from string **z** starting at the position a with a length of **length**. Example:<br>SELECT SUBSTRING('wardrobe',1,4) = 'ward' |
| UNICODE | Returns the integer value, as defined by the Unicode standard, for the first character of the input expression. |
| UPPER(z) | Converts all lowercase letters of string **z** to uppercase letters. Uppercase letters and numbers do not change. Example:<br>SELECT UPPER('loWer') = 'LOWER' |

## System Functions

System functions of Transact-SQL provide extensive information about database objects. Most system functions use an internal numeric identifier (ID), which is assigned to each database object by the system at its creation. Using this identifier, the system can uniquely identify each database object. System functions provide information about the database system. The following table describes several system functions. (For the complete list of all system functions, please see Books Online.)

| Function | Explanation |
|---|---|
| CAST(a AS type [(length)] | Converts an expression **a** into the specified data type **type** (if possible). **a** could be any valid expression. Example:<br>SELECT CAST(3000000000 AS BIGINT) = 3000000000 |
| COALESCE($a_1$,$a_2$,...) | Returns for a given list of expressions $\mathbf{a_1}$, $\mathbf{a_2}$,... the value of the first expression that is not NULL. |
| COL_LENGTH(obj,col) | Returns the length of the column **col** belonging to the database object (table or view) **obj**. Example:<br>SELECT COL_LENGTH('customers', 'cust_ID') = 10 |
| CONVERT(type[(length)],a) | Equivalent to CAST, but the arguments are specified differently. CONVERT can be used with any data type. |
| CURRENT_TIMESTAMP | Returns the current date and time. Example:<br>SELECT CURRENT_TIMESTAMP = '2008-01-01 17:22:55.670' |
| CURRENT_USER | Returns the name of the current user. |
| DATALENGTH(z) | Calculates the length (in bytes) of the result of the expression **z**. Example:<br>SELECT DATALENGTH(ProductName) FROM products. (This query returns the length of each field.) |
| GETANSINULL('dbname') | Returns 1 if the use of NULL values in the database **dbname** complies with the ANSI SQL standard. (See also the explanation of NULL values at the end of this chapter.) Example:<br>SELECT GETANSINULL('AdventureWorks') = 1 |
| ISNULL(expr, value) | Returns the value of **expr** if that value is not null; otherwise, it returns **value** (see Example 5.22). |
| ISNUMERIC(expression) | Determines whether an expression is a valid numeric type. |
| NEWID() | Creates a unique ID number that consists of a 16-byte binary string intended to store values of the UNIQUEIDENTIFIER data type. |
| NEWSEQUENTIALID() | Creates a GUID that is greater than any GUID previously generated by this function on a specified computer. (This function can only be used as a default value for a column.) |
| NULLIF($expr_1$,$expr_2$) | Returns the NULL value if the expressions $\mathbf{expr_1}$ and $\mathbf{expr_2}$ are equal. Example:<br>SELECT NULLIF(project_no, 'p1') FROM projects. (The query returns NULL for the project with the project_no = 'p1'). |
| SERVERPROPERTY(propertyname) | Returns the property information about the database server. |
| SYSTEM_USER | Returns the login ID of the current user. Example:<br>SELECT SYSTEM_USER = LTB13942\dusan |
| USER_ID([user_name]) | Returns the identifier of the user **user_name**. If no name is specified, the identifier of the current user is retrieved. Example:<br>SELECT USER_ID('guest') = 2 |
| USER_NAME([id]) | Returns the name of the user with the identifier **id**. If no name is specified, the name of the current user is retrieved. Example:<br>SELECT USER_NAME = 'guest' |

All string functions can be nested in any order; for example, REVERSE (CURRENT_USER).

## Metadata Functions

Generally, metadata functions return information concerning the specified database and database objects. The following table describes several metadata functions. (For the complete list of all metadata functions, please see Books Online.)

| Function | Explanation |
|---|---|
| COL_NAME(tab_id, col_id) | Returns the name of a column belonging to the table with the ID **tab_id** and column ID **col_id**. Example: SELECT COL_NAME(OBJECT_ID('employee') , 3) = 'emp_lname' |
| COLUMNPROPERTY (id, col, property) | Returns the information about the specified column. Example: SELECT COLUMNPROPERTY(object_id('project'), 'project_no', 'PRECISION') = 4 |
| DATABASEPROPERTY(database, property) | Returns the named database property value for the specified database and property. Example: SELECT DATABASEPROPERTY('sample', 'IsNullConcat') = 0. (The **IsNullConcat** property corresponds to the option CONCAT_NULL_YIELDS_NULL, which is described at the end of this chapter.) |
| DB_ID([db_name]) | Returns the identifier of the database **db_name**. If no name is specified, the identifier of the current database is returned. Example: SELECT DB_ID('AdventureWorks') = 6 |
| DB_NAME([db_id]) | Returns the name of the database with the identifier **db_id**. If no identifier is specified, the name of the current database is displayed. Example: SELECT DB_NAME(6) = 'AdventureWorks' |
| INDEX_COL(table, i, no) | Returns the name of the indexed column in the table **table**, defined by the index identifier **i** and the position **no** of the column in the index. |
| INDEXPROPERTY(obj_id, index_name, property) | Returns the named index or statistics property value of a specified table identification number, index or statistics name, and property name. |
| OBJECT_NAME(obj_id) | Returns the name of the database object with the identifier **obj_id**. Example: SELECT OBJECT_NAME(453576654) = 'products' |
| OBJECT_ID(obj_name) | Returns the identifier of the database object **obj_name**. Example: SELECT OBJECT_ID('products') = 453576654 |
| OBJECTPROPERTY(obj_id,property) | Returns the information about the objects from the current database. |

## Scalar Operators

Scalar operators are used for operations with scalar values. Transact-SQL supports numeric and Boolean operators as well as concatenation.

There are unary and binary arithmetic operators. Unary operators are + and – (as signs). Binary arithmetic operators are +, –, *, /, and **%**. (The first four binary operators have their respective mathematical meanings, whereas **%** is the modulo operator.)

Boolean operators have two different notations depending on whether they are applied to bit strings or to other data types. The operators NOT, AND, and OR are applied to all data types (except BIT). They are described in detail in Chapter 6.

The bitwise operators for manipulating bit strings are listed here, and Example 4.8 shows how they are used:

- ~ Complement (i.e., NOT)
- **&** Conjunction of bit strings (i.e., AND)
- | Disjunction of bit strings (i.e., OR)
- ^ Exclusive disjunction (i.e., XOR or Exclusive OR)

**EXAMPLE 4.8**

```
~(1001001) = (0110110)
(11001001) | (10101101) = (11101101)
(11001001) & (10101101) = (10001001)
(11001001) ^ (10101101) = (01100100)
```

The concatenation operator + can be used to concatenate two character strings or bit strings.

## Global Variables

Global variables are special system variables that can be used as if they were scalar constants. Transact-SQL supports many global variables, which have to be preceded by the prefix **@@**. The following table describes several global variables. (For the complete list of all global variables, see Books Online.)

| Variable | Explanation |
|---|---|
| @@CONNECTIONS | Returns the number of login attempts since starting the system. |
| @@CPU_BUSY | Returns the total CPU time (in units of milliseconds) used since starting the system. |
| @@ERROR | Returns the information about the return value of the last executed Transact-SQL statement. |

| Variable | Explanation |
|---|---|
| @@IDENTITY | Returns the last inserted value for the column with the IDENTITY property (see Chapter 6). |
| @@LANGID | Returns the identifier of the language that is currently used by the database system. |
| @@LANGUAGE | Returns the name of the language that is currently used by the database system. |
| @@MAX_CONNECTIONS | Returns the maximum number of actual connections to the system. |
| @@PROCID | Returns the identifier for the stored procedure currently being executed. |
| @@ROWCOUNT | Returns the number of rows that have been affected by the last Transact-SQL statement executed by the system. |
| @@SERVERNAME | Retrieves the information concerning the local database server. This information contains, among other things, the name of the server and the name of the instance. |
| @@SPID | Returns the identifier of the server process. |
| @@VERSION | Returns the current version of the database system software. |

# NULL Values

A NULL value is a special value that may be assigned to a column. This value is normally used when information in a column is unknown or not applicable. For example, in the case of an unknown home telephone number for a company's employee, it is recommended that the NULL value be assigned to the **home_telephone** column.

Any arithmetic expression results in a NULL if any operand of that expression is itself a NULL value. Therefore, in unary arithmetic expressions (if A is an expression with a NULL value), both +A and –A return NULL. In binary expressions, if one (or both) of the operands A or B has the NULL value, A + B, A – B, A * B, A / B, and A % B also result in a NULL. (The operands A and B have to be numerical expressions.)

If an expression contains a relational operation and one (or both) of the operands has (have) the NULL value, the result of this operation will be NULL. Hence, each of the expressions A = B, A <> B, A < B, and A > B also returns NULL.

In the Boolean AND, OR, and NOT, the behavior of the NULL values is specified by the following truth tables, where T stands for true, U for unknown (NULL), and F for false. In these tables, follow the row and column represented by the values of the Boolean expressions that the operator works on, and the value where they intersect represents the resulting value.

| AND | T | U | F |
|---|---|---|---|
| T | T | U | F |
| U | U | U | F |
| F | F | F | F |

| OR | T | U | F |
|---|---|---|---|
| T | T | T | T |
| U | T | U | U |
| F | T | U | F |

| NOT | |
|---|---|
| T | F |
| U | U |
| F | T |

Any NULL value in the argument of aggregate functions AVG, SUM, MAX, MIN, and COUNT is eliminated before the respective function is calculated (except for the function COUNT(*)). If a column contains only NULL values, the function returns NULL. The aggregate function COUNT(*) handles all NULL values the same as non-NULL values. If the column contains only NULL values, the result of the function COUNT(DISTINCT column_name) is 0.

A NULL value has to be different from all other values. For numeric data types, there is a distinction between the value zero and NULL. The same is true for the empty string and NULL for character data types.

A column of a table allows NULL values if its definition explicitly contains NULL. On the other hand, NULL values are not permitted if the definition of a column explicitly contains NOT NULL. If the user does not specify NULL or NOT NULL for a column with a data type (except TIMESTAMP), the following values are assigned:

- **NULL** If the ANSI_NULL_DFLT_ON option of the SET statement is set to ON
- **NOT NULL** If the ANSI_NULL_DFLT_OFF option of the SET statement is set to ON

If the SET statement isn't activated, a column will contain the value NOT NULL by default. (The columns of TIMESTAMP data type can only be declared as NOT NULL columns.)

There is also another option of the SET statement: CONCAT_NULL_YIELDS_NULL. This option influences the concatenation operation with a NULL value so that anything you concatenate to a NULL value will yield NULL again. For instance:

'San Francisco' + NULL = NULL

## Conclusion

The basic features of Transact-SQL consist of data types, predicates, and functions. Data types comply with data types of the ANSI SQL92 standard. Transact-SQL supports a variety of useful system functions.

The next chapter introduces you to Transact-SQL statements in relation to SQL's data definition language. This part of Transact-SQL comprises all the statements needed for creating, altering, and removing database objects.

## Exercises

**E.4.1**

What is the difference between the numeric data types INT, SMALLINT, and TINYINT?

### E.4.2

What is the difference between the data types CHAR and VARCHAR? When should you use the latter (instead of the former) and vice versa?

### E.4.3

How can you set the format of a column with the DATE data type so that its values can be entered in the form 'yyyy/mm/dd'?

In the following two exercises, use the SELECT statement in the Query Editor component window of SQL Server Management Studio to display the result of all system functions and global variables. (For instance, SELECT host_id() displays the ID number of the current host.)

### E.4.4

Using system functions, find the ID number of the **test** database (Exercise 2.1).

### E.4.5

Using the system variables, display the current version of the database system software and the language that is used by this software.

### E.4.6

Using the bitwise operators **&**, |, and ^, calculate the following operations with the bit strings:

(11100101) & (01010111)
(10011011) | (11001001)
(10110111) ^ (10110001)

### E.4.7

What is the result of the following expressions? (A is a numerical and B a logical expression.)

A + NULL
NULL = NULL
B OR NULL
B AND NULL

### E.4.8

When can you use both single and double quotation marks to define string and temporal constants?

### E.4.9

What is a delimited identifier and when do you need it?